Chrysostome BASKIOTIS

SCILAB

EISTI, 2006

Chapitre 1

INTRODUCTION AU SCILAB

Scilab est un environnement de programmation de haut niveau développé par l'INRIA et distribué en "Open Source Software" et son utilisation est libre. Son site officiel est `http ://scilabsoft.inria.fr/`. La dernière version est la version 7.2 qui

dispose en plus d'un éditeur intégré.

Scilab permet de faire de façon relativement facile des calculs numériques. Il permet aussi de faire de graphiques en deux et trois dimensions et exporter ces graphiques en format postscript. De plus il permet de manipuler des images et des sons. Il est possible, par des commandes appropriées, de mettre les résultats des calculs effectuées par Scilab, sous format LaTeX. De plus Scilab dispose des nombreuses boîtes à outils dans des domaines variés comme l'optimisation, les réseaux des neurones, l'automatique, les éléments finis, etc.

1.1. Introduction

On peut utiliser Scilab de deux façons :

- soit comme une calculatrice scientifique, en écrivant les instructions directement sur l'espace de travail ;
- soit comme un langage de programmation qui exécute des instructions enregistrées, à l'aide d'un éditeur des textes, sur un fichier.

Le langage de programmation est le même dans les deux cas. Nous donnons dans la suite les commandes principales de Scilab.

1.1.1. Affichage

Une instruction Scilab se termine par " ;". Si on omet ce symbole, Scilab affiche sur l'écran le résultat de l'opération précédé de l'indication `ans =`. Pour l'affichage, Scilab dispose par défaut de 10 places pour afficher une valeur numérique. Ainsi si on tape :

```
-->1/8.1000000729000057
```

on obtient comme réponse :

```
ans =
.1234568
```

c'est-à-dire 7 chiffres après la décimale auquels se rajoutent le point décimal, le 0 qui n'apparaît pas puisqu'il est 0, et le signe + qui n'apparaît pas non plus puisqu'il +, donc au total 10.

Nous constatons que la "," des nombres décimaux est remplacée par le ".".

Si nous voulons augmenter le nombre de places pour l'affichage, on doit utiliser la commande :

```
->format("v",19);
```

qui permet d'afficher 16 chiffres. Ainsi nous avons :

```
->1/8.1000000729000057

 ans =

 .1234567890123456
```

Le format "v" signifie que le format utilisé pour l'affichage sera le format avec virgule décimale. Si nous voulons avoir un affichage avec des exposants de 10, le format à utiliser est "e". Ainsi :

```
->format("e",22);

->1/8.1000000729000057

 ans =

 1.2345678901234 56E-01
```

fournit aussi un affichage avec 16 places.

Il est à noter que si plusieurs instructions de `format` se succèdent, seule la dernière instruction est prise en compte.

Scilab reconnaît la notation scientifique :

```
->1.2E-1

 ans =
```

```
        0.12
```

ou encore

```
        ->1.2d-1

         ans =

         0.12
```

1.1.2. Variables prédéfinies

Scilab comporte un certain nombre de variables prédéfinies. Le nom d'une variable prédéfinie commence par le symbole "%". La table suivante fournit la liste de ces variables.

Variable	Signification
`% pi`	La constante $\pi = 3.14159265358979310$
`% i`	$\sqrt{-1} = j$
`% e` hline `% eps`	La constate $e = 3.14159265358979310$ La précision machine $eps = 2.22E-16 = .0000000000000000222$ pour Intelx86 double précision.
`% nan`	Not a number
`% inf`	Infini
`%t` ou `%T`	Vrai
`%f` ou `%F`	Faux
`%io(1)` et `%io(2)`	Ils désignent les numéros logiques des fichiers en entrée (clavier) et en sortie (écran) respectivement

Par exemple le nombre complexe $a + jb$ s'écrit en Scilab `a + %i * b`.

1.1.3. Opérations arthmétiques

Nous avons les opérations arithmétiques usuelles plus la division gauche.

Symbole	Opération
+	Addition
-	Soustraction
*	Multiplication
/	Division
\	Division à gauche
^ou **	Exponentiation

Comparons division et division à gauche :

```
->4/2

 ans =

 2.

->4\2

 ans =

 .5
```

1.1.4. Fonctions élémentaires

Scilab dispose d'un certain nombre de fonctions standards comme par exemple : `sqrt, log10, cos`, etc.

Pour avoir la liste il faut, dans la fenêtre de Scilab, cliquer sur Help et ensuite sur le menu déroulant à Elementary Functions.

Nous donnons ci-après queques unes de ces fonctions.

– Fonctions logarithmiques :

Nom fonction	Logarithme	Exemple
`log(x)`	log népérien	`->log(10)`
		`ans =`
		`2.3025850929940459`
`log10(x)`	log décimal	`->log(10)`
		`ans =`
		`1.`
`log2(x)`	log binaire	`->log(10)`
		`ans =`
		`3.3219280948873626`

– Opérations arithmétiques symboliques

Nom fonction	Opération symbolique	Exemple

`addf(a,b)`	addition symbolique de `a` et `b`	`->addf("a","b")`
		`ans =`
		`a+b`

`subf(a,b)`	soustraction symbolique de `a` et `b`	`->addf("a","b")`
		`ans =`
		`a-b`

`mulf(a,b)`	multiplication symbolique de `a` et `b`	`->mulf("a","b")`
		`ans =`
		`a*b`

`rdivf(a,b)`	division symbolique de `a` par `b`	`->rdivf("a","b")`
		`ans =`
		`a/b`

`ldivf(a,b)`	division symbolique de `b` par `a`	`->ldivf("a","b")`
		`ans =`
		`b/a`

`eval(c)`	évaluation de la valeur
	de la variable symbolique `c`
		`->a = 1, b = 2,addf("a","b"),`
		`d = eval(c)`
		`ans =`
		`a = 1, b = 2, c = a+b, d = 3`

`eval("c")`	évaluation de la valeur symbolique
	de la variable symbolique `c`
		`->a = 1, b = 2, addf("a","b"),`
		`d = eval("c")`
		`ans =`
		`a = 1, b = 2, c = a+b, d = a+b`

– Partie entière d'une valeur réelle :

Nom fonction	Valeur de la fonction	Exemple

`ceil(x)`	Partie entière par excès	`->ceil(2.4) , ceil(-2.4)`
		`ans =`
		`3., -2.`

`floor(x)`	Partie entière	`->floor(2.4) , floor(-2.4)`
		`ans =`
		`2., -3.`

`fix(x)` ou `int(x)`	Partie entière la plus proche de zéro	`->fix(2.4)` ou `int(2.4), fix(-2.4)`
		`ans =`
		`2., -2.`

`round(x)`	Arrondi à la partie entière la plus proche de la valeur donnée.	`->round(2.4) , round(-2.9)`
		`2., -3.`
		`ans =`
		`2., -3.`

– Conversion des réels en entiers :

Nom fonction	Valeur de la fonction	Exemple
`iconvert(x,itype)`	Conversion de x en un nombre selon la valeur de `itype` `itype = 0` ⇒ conversion en nombre réel	`iconvert(123.45678901234`
	`itype = 1` ⇒ conversion en entier stocké dans 1 octet. Valeur : $[-128, 127]$	`iconvert(123.45678901234`
	`itype = 2` ⇒ conversion en entier stocké dans 2 octets. Valeur : $[-32768, 327677]$	`iconvert(12345.678901234`
	`itype = 4` ⇒ conversion en entier stocké dans 4 octets. Valeur : $[-2147483648, 2147486447]$	`iconvert(1234567890.1234`
	`itype = 11` ⇒ conversion en entier non signé stocké dans 1 octet. Valeur : $[0, 255]$	`iconvert(123.45678901234`
	`itype = 12` ⇒ conversion en entier non signé stocké dans 2 octets. Valeur : $[0, 65535]$	`iconvert(123456.78901234`
	`itype = 14` ⇒ conversion en entier non signé stocké dans 4 octets. Valeur : $[0, 4294967295]$	`iconvert(1234567890.1234`
double(x)	Conversion d'un entier en réel double précision	`double(1234567890)`

1.2. Vecteurs et matrices

La matrice $\mathbf{A} = \begin{bmatrix} 2 & 4 & 6 \\ 3 & 5 & 7 \end{bmatrix}$ sera introduite dans Scilab de la façon suivante :

```
-> A=[2, 4, 6; 3, 5, 7]
```

c'est-à-dire les virgules séparent les éléments de la même ligne et les points-virgules les colonnes.

Scilab est un langage de programmation matriciel. C'est-à-dire son élément de base n'est pas le nombre scalaire `x` comme avec e.g. le Pascal ou le C ou le Fortran, mais la matrice `[x]`, qui représente ici le scalaire `x`.

De ce fait les opérations arithmétiques usuelles – à l'exception de la division – et les fonctions élémentaires s'appliquent aux vecteurs et matrices. En plus nous avons aussi un nouveau type de multiplication qui est la multiplication de deux matrices termes par terme. Ainsi, étant données deux matrices de même format `A` et `B`, on a la multiplication terme par terme `C=A.*B` définie par `c(i,j) = a(i,j) * b(i,j)`. Il y a une extension de la multiplication terme par terme à la division et à la division à gauche terme par terme. Ainsi `C=A./B` définie par `c(i,j) = a(i,j) / b(i,j)` et `C=A.\B` définie par `c(i,j) = b(i,j) / a(i,j)`.

En ce qui concerne l'exponentiation nous avons :

- `[a, b] ^2 = [a`^{2}`, b`^{2}`]`
- `[a, b] .^2 = [a`^{2}`, b`^{2}`]`
- `[a, b; c, d]^2 =[a`^{2}`+bc, ab+bd; ca+dc, cb+d`^{2}`]`
- `[a, b; c, d].^2 =[a`^{2}`, b`^{2}`, c`^{2}`, d`^{2}`]`

Le tableau suivant fournit quelques commandes spécifiques pour la création des vecteurs et matrices.

Fonction	Signification	Exemple

`eye(m,n)`	Création d'une matrice identité	`->eye(2,3)`
		`ans =`
		`! 1. 0. 0. !`
		`! 0. 1. 0. !`

`ones(m,n)`	Création d'une matrice remplie de 1	`->ones(2,3)`
		`ans =`
		`! 1. 1. 1. !`
		`! 1. 1. 1. !`

`zeros(m,n)`	Création d'une matrice nulle	`->zeros(2,3)`

La prise en compte des dimensions des matrices se fait par les fonctions :

Fonction	Signification	Exemple

`[m, n] = size(a)`	Nombre de lignes et de colonnes de la matrice `a`	`->[m, n] = size(eye(2,3))`
		`ans =`
		`n = 3, m = 2`

`length(a)`	Nombre de cases de la matrice `a`	`->length(eye(2,3))`
		`ans =`
		`6`

Le tri des éléments d'une matrice se fait par la fonction `sort` :

Fonction	Signification	Exemple

`a = rand(m,n)`	`a` est une matrice aléatoire de dimensions `m` et `n`	`->a= rand(2,3)`
		`ans =`
		`! .211324865 .000221135 .665`
		`! .756043854 .330327092 .628`

`b = sort(a)`	La matrice `b` est ordonnée selon les valeurs décroissantes de `a` rangées colonne par colonne	`->sort(a)`
		`ans =`
		`! .756043854 .628391788 .211`
		`! .665381104 .330327092 .000`

`b = sort(a,'c')`	Chaque ligne de la matrice est ordonné selon les valeurs décroisantes de ses éléments	`->sort(a,'c')`
		`ans =`
		`! .665381104 .211324865 .000`
		`! .756043854 .628391788 .330`

`b = sort(a,'r'`	Chaque ligne de la matrice est ordonnée selon les valeurs décroisantes de ses éléments	`->sort(a,'c')`
		`ans =`
		`! .756043854 .330327092 .665`
		`! .756043854 .330327092 .665`

1.3. Programmation

En règle générale pour des questions de rapidité d'exécution en Scilab il faut privilégier les opérations vectorielles et matricielles. Pour la programmation nous disposons d'un ensemble d'opérateurs de comparaison et de logique qui sont présentés à la table suivante :

Opérateurs de comparaison	Signification
<	plus petit
>	plus grand
<=	plus petit ou égal
>=	plus grand ou égal
<>	différent
==	test d'égalité
Opérateurs logiques	**Signification**
&	et
|	ou
~	non

1.3.1. Blocs d'itérativité et de contôle

Les boucles en `SciLab`.– En programmation nous utilisons les boucles pour effectuer des calculs qui sont repétitifs. Par exemple si nous voulons faire la somme de tous les entiers entre 5 et 14 nous pouvons écrire le programme `SciLab`

```
somme = 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 +14;
```

mais nous pouvons aussi écrire le programme

```
somme = 0;
```

```
for i = 5 :14

    somme = somme + i;

end;
```

Dans cet exemple `i` est l'indice de progression de la boucle, `5` est sa première valeur et `14` sa dernière valeur.

Les avantages de ce programme sont multiples :

- Flexibilité : Le même programme peut être utiliser si nous voulons faire la somme de -5 à 14 ou de 14 à 5. Il suufit de paramétrer les valeurs du début et de la fin de la boucle. Par exemple pour sommer de 14 à 5 on peut écrire :
  ```
  somme = 0;
  indFin = 14;
  indDebut = 5;
  for i = indFin :-1 :indDebut
      somme = somme + i;
  end;
  ```
- Maintenabilité : Toute modification des limites de la boucle se fait en changeant les valeurs de `indDebut` et `indFin`. Nous réduisons ainsi le nombre d'erreurs qu'il soit possible de faire lors d'une mise à jour du programme.

L'erreur la plus fréquente est d'oublier d'initialiser la valeur de la variable `somme`. Si `somme` n'a pas été utilisée avant, `SciLab` l'indiquera. Mais si `somme` a déjà été utilisée, nous n'aurons aucun avertissement et le résultat sera faux. Une autre erreur, un peu moins fréquente, est d'utiliser l'indice de la progression de la boucle avec d'autres affectations à l'intérieur de la boucle. Bien sûr dans ce cas le nombre d'itérations de la boucle se trouve alterée.

Les instructions de branchement conditionnel en `SciLab`.– Un programme normalement se déroule de façon linéaire. si nous voulons changer l'ordre du déroulement des instructions nous devons utiliser une instructions de branchement conditionnel. En `SciLab` il y en a trois.

(1) L'instruction `if` : Elle permet, en fonction de la valeur d'une expression – appelée condition, d'effectuer ou non un ensemble d'instructions.Sa syntaxe est la suivante

```
if <condition1> then <actions1>
```

```
[elseif <condition2> then <action2>]
....
[else <actions>]
end;
```

L'instruction `if` évalue la `<condition>` et si elle est vérifiée, alors elle exécute le groupe d'instructions qui sont dans le bloc`<actions>`. L'instruction optionnelle `elseif` fournit la possibilité d'exécuter un ensemble d'instructions selon d'autres conditions. L'instruction optionnelle `elseif` est utilisée si nous avons à faire une action par défaut.

L'erreur la plus fréquente est d'utiliser de façon mauvaise les conditions. En général il faut éviter la succession des plusieurs `elseif`.

(2) L'instruction `select` : Elle permet en fonction de la valeur d'une variable, d'effectuer ou non un ensemble d'instructions. Sa syntaxe est la suivante

```
select variable,
case valeur1 then <actions1>,
case valeur2 then <action2>,
...
case valeurn then <actionsn>,
[else <actions>],
end;
```

Le fonctionnement de `select` est analogue à celui de `if` à la seule différence que la condition porte sur une variable et non pas sur une expression complexe.

L'erreur la plus fréquente est que la variable ne prend jamais une valeur particulière ou, au contraire, elle prend une valeur qui n'est pas repertoriée.

(3) L'instruction `while` : Elle permet de contrôler une boucle par la valeur d'une autre variable que l'indice de progression de la boucle. Sa syntaxe est la suivante

```
while <condition>
    <actions>
end;
```

Tant que la `<condition>` est vérifiée, les `<actions>` de la boucle sont exécutées.

Le maniement de cette instruction est particulièrement délicat. Il faut faire attention à la nature de la condition et au fait que si la condition n'est pas vérifiée la boucle n'est pas exécutée même pas une fois. (À ne pas confondre avec l'instruction `repeat` qu'elle n'existe pas en `SciLab`).

1.3.2. Fonctions

La notion de fonction en Scilab regroupe les notions de fonction et de procédure des autres langages de programmation. Une fonction doit être stockée dans un fichier qui porte l'extension sci. Dans un même fichier peuvent être stockées plusieurs fonctions différentes. L'invocation d'un fichier des fonctions se fait à l'aide de l'instruction

```
getf('nomFichierFonctions.sci');
```

La syntaxe d'une fonction est la suivante :

```
function [out1,out2,...,outN]=nomFonction(in1,in2,...,inM)


// out1,out2,...,outN sont les variables de sortie


// in1,in2,...,inM variables d' entree

 < instructions >

endfunction
```

L'appel à cette fonction se fait comme suit :

```
[out1,out2,...,outN]=nomFonction(in1,in2,...,inM);
```

Une facilité offerte par Scilab est de ne pas utiliser toutes les variables en entrée ou en sortie lors des appels à une fonction. Si par exemple nous avons besoin de la 1e sortie te de la 1e et 3e entrée pour un usage particulier de la fonction, nous pouvons utiliser l'appel suivant :

```
[out1]=nomFonction(in1,in2,in3);
```

Scilab reconnaît des variables globales et locales. Toutes les variables qui n'apparaissent que dans des fonctions sont des variables locales, sauf si elles sont déclarés globales par la commande

```
global nomVariable
```

Les variables qui apparaissent dans le programme principal sont des variables globales. En tant que telles peuvent être utilisées par des fonctions mais elle ne peuvent pas être modifiées par ces fonctions. Si nous voulons qu'une variable soit modifiée par une fonction, il faut qu'elle apparaisse explicitement dans la liste des variables en sortie. De toute manière une bonne habitude à prendre est de ne pas utiliser les facilitées offertes par les langages de programmation et de faire apparaître toutes les variables utilisées et/ou modifiées dans la liste des variables en entrée et/ou en sortie.

1.3.3. Primitives utilisées par les fonctions

La récupération d'une erreur se fait à l'aide de la primitive `error` qui permet d'afficher un message d'erreur et d'arrêter le deroulement de la fonction

Exemple : `error('erreur dans la fonction');`

Nous pouvons arrêter l'exécution de la fonction et revenir au programme appelant en utilisant la primitive `return` :

Exemple :

```
// division z = x/y

    if (y == 0) then

        z = %inf;

        return;

    end

    z = x/y;
```

Si on veut qu'un message d'erreur apparaît à l'écran sans que l'exécution de la fonction soit arrêter, on peut utiliser la primitive `warning` selon l'exemple suivant :

Exemple : `warning('erreur dans la fonction');`

Pour connaître le nombre d'arguments utilisées effectivement lors de l'appel d'une fonction on peut avoir la primitive `argn` comme suit :

Exemple : `[lhs, rhs] = argn(0);`
où `lhs` fournit le nombre d'argument en sortie effectifs et `rhs` le nombre d'arguments en entrée effectifs.

1.4. Graphiques

Scilab possède plusieurs fonctions qui permettent de faire plusieurs types de graphiques de manière relativement simple.

Pour afficher un graphique il faut d'abord ouvrir une fenêtre graphique. Le tableau suivant fournit les primitives à utiliser pour ouverture d'une fenêtre graphique.

Primitive	Fonction
`xset("window",num)`	Création et ouverture de la fenêtre graphique numéro `num` qui devient la fenêtre active
`xselect()`	Active la fenêtre graphique courante
`xbasc([num])`	Efface la fenêtre graphique numéro `num`
`xdel([num])`	Supprime la fenêtre graphique numéro `num`

Scilab peut gérer plusieurs fenêtres en même temps mais à chaque instant une seule fenêtre, dite *fenêtre courante*, est active. À chaque fenêtre est asocié un contexte défini par la primitive `xset`. Pour voir les différents contextes qui existent se reporter à l'aide de Scilab.

1.4.1. Courbes à 2D

Supposons que nous avon `m` courbes à tracer, chacune ayant `n` points. On construit les deux tableaux `X` et `Y` de format `n×m`. La primitive pour le tracé de ces courbes est la suivante :

```
plot2d(X, Y, style, "xyz", leg, [xmin,ymin,xmax,ymax], [nx,Nx,ny,Ny])
```

avec

- `style` : un tableau ligne de m dimensions, chaque élément représente le style du tracé de la courbe correspondante. Si le style a une valeur positive, alors cette valeur indique la couleur de la courbe et le tracé sera continu. Si la valeur est non positive, alors cette valeur indique le symbole qui sera utilisé pour tracé les points de la courbe et le tracé sera discontinu. Les tableaux suivants donnent la correspondance des couleurs et des symboles avec les valeurs.
  numéro couleur

Valeur	Couleur
1	noire
2	bleue
3	verte
4	azur
5	rouge
6	magenta
26	marron
29	rose
32	jaune

Valeur	Symbole
0	·
-1	+
-2	×
-3	⊕
-4	♦
-5	◇
-6	▽
-7	△
-8	♣
-9	◯

- `"xyz"` . La valeur `x` est relative à l'affichage des légendes des courbes. Si `x=1`, alors il y a affichage du titre de chaque courbe.
  La valeur de `y` fournit le mode de calcul de l'échelle. Si `y=0`, l'échelle précédente est utilisée. Si `y=1`, l'échelle fixée par le programmeur est utilisée. Si `y=2`, Scilab calcule au mieux l'échelle.
  La valeur de `z` règle le cadre du graphique. Si `z=0`, il n'y a pas de cadre. Si `z=1`, le graphique est entouré d'un cadre gradué. Si `z=2`, le graphique est entouré d'un cadre.
- `leg` est la légende des courbes qui doit être présente si `x=1`. Les légendes des différentes courbes doivent être séparées par `@`.
- `[xmin,ymin,xmax,ymax]` fournit l'échelle si `y=1`.
- `[nx,Nx,ny,Ny]` indique les graduations dans le cas où `z=1`. `Nx` est le nombre d'intervalles en `x` et `nx` est le nombre de

sous-intervalles par intervalle.

1.4.2. Surfaces et courbes à 3D

Comme pour les graphiques en 2D, pour tracer une surface en 3D on utilise la primitive correspondante `plot3D`. Si par exemple nous avons la surface $z(x,y) = f(x,y)$ à tracer, on discrétise x et y en `nx` et `ny` points respectivement et on utilise la primitive

```
plot3d(x, y, z, theta, alpha, leg, [mode type box], )
```

avec :

- `x` et `y` sont deux vecteurs-ligne de format `(1×nx)` et `(1×ny)` qui correspondent à la discrétisation en `x` et en `y`.
- `z` est un tableau de dimension `(nx×ny)` telle que `z(i,j) = f(x(i),y(j))`.
- `theta` et `alpha` sont les valeurs en degrés de deux angles qui déterminent le point de vue de la surface.
- `leg` est la légende pour les axes (par exemple `leg="x@y@z"`).
- `"xyz"`. La valeur de `x` est relative à la présence de faces cachées et du maillage. Si `x < 0`, alors le maillage et les faces cahées sont supprimés. Si `x = 0`, le maillage et les faces cachées sont présents. Si `x > 0`, les faces cachées sont supprimées.
  La valeur de `y` joue le même rôle qu'en `plot2d`.
  La valeur de `z` règle le cadre et les axes du graphique. Si `z=0`, il n'y a pas de cadre. Si `z=2`, les axes sont dessinés ? Si `z=3`, un cadre est dessiné. Si `z=4`, un cadre et les axes sont dessinés.
- `[xmin,xmax,ymin,ymax,zmin,zmax]` fournit l'échelle si `y=1`.

Si on veut que les facettes aient des couleurs différentes selon les valeurs de `z(i,j)` , alors il faut utiliser `plot3D1` avec les mêmes arguments.

Si on veut tracer une courbe en 3D, il faut utiliser la primitive `param3D` qui a comme sequence d'appel :

```
param3d(x, y, z, theta, alpha, leg, [type box], [xmin,xmax,ymin,ymax,zmin,zmax])
```

avec la même signification qu'en `plot3D`.

Si nous voulons tracer plusieurs courbes sur le même graphique, on doit utiliser `param3D1`.

1.4.3. Stocker les graphiques sur des fichiers externes

Nous avons la possibilité de stocker une fenêtre graphique sur un fichier en format soit postscript, soit gif, soit xfig. Pour cela il faut

(1) sur le menu File de la fenêtre graphique, cliquer sur l'item Export ;

(2) utiliser la fenêtre qui apparaît pour choisir le format du fichier (postscript, gif ou xfig), l'orientation (portrait, paysage) et la couleur ;

(3) donner le nom du fichier.

1.5. Fichiers en Scilab

Nous examinerons dans cette section la création, l'écriture et la lecture des fichiers en Scilab. Étant donné que le clavier et l'écran sont aussi des fichiers, nous verrons, en même temps, la lecture des données au clavier et l'affichage à l'écran.

1.5.1. Données non formatées

Pour afficher la valeur d'une variable `x` nous avons deux possibilités très simples :

- `disp(x,'x = ');`
- `print(%io(2), x, 'x = ');` ou `print(6, x, 'x = ');`

Il s'agit d'affichages rudimentaires sans possibilité de présentation particulière de l'affichage.

La primitive `print` fait référence au nom du fichier qui est `%io(2)` ou le numéro `6` pour l'écran. Par extension nous pouvons envisager d'utiliser `print` pour des fichiers texte, comme par exemple

```
print('donnees.txt', x, 'x = ');
```

Notons que ce fichier peut être, par la suite, lu par un éditeur des textes.

Si nous voulons lire par Scilab le fichier des données que nous avons sauvegarder, il faut utiliser pour la sauvegarde, la primitive save :

```
save('donnees.dat',x,y,z);
```

qui stocke dans le fichier `donnees.dat` le contenu de `x, y` et `z`.

Pour la lecture on utilise la primitive load :

```
load('donnees.dat',x,y,z);
```

Le fichier créé est un fichier binaire et donc n'est pas utilisable avec un éditeur de texte. Mais, contrairement aux fichiers binaires classiques, nous ne pouvons avoir plusieurs enregistrements. Chaque fois qu'on écrit sur ce fichier, on detruit l'enregistrement précédent.

Nous pouvons lire de façon interactive des données rentrés au clavier à l'aide de la primitive input :

```
[x]=input("Donner la valeur de x : ", "string")
```

où `x` est la variable dans laquelle sera stockée la donnée, `"string"` ou `"s"` est présent si la donnée est une chaîne des caractères.

1.5.2. Données formatées

Nous donnons ci-après les opérations sur des fichiers selon une approche inspirée de la gestion des fichiers en Fortran.

Pour écrire ou lire sur un fichier, il faut d'abord l'ouvrit :

```
[nomUnite, err] = file('open', nomFichier, statut, typeAcces, recBytes, typeFormat);
```

avec

- `nomUnite` : Le nom logique du fichier. Notons que pour le clavier nous avons `%io(1)` et pour l'écran `%io(2)`.
- `err` : Numéro d'erreur. Si `err = 0`, pas d'erreur.
- `nomFichier` : Nom du fichier.
- `statut` : Nature du fichier, à savoir "`new`" nouveau (nature par défaut), "`old`" ancien, "`unknow`" non précisé et "`scratch`" à détruire à la fin de la session.
- `typeAcces` : Type d'accès au fichier, à savoir "`sequential`" pour séquentiel (type par défaut) et "`direct`" pour accès direct.
- `recBytes` : Nombre d'octets par enregistrement dans le cas d'accès direct.
- `typeFormat` : "`formatted`" pour des données formatées (type par défaut) et "`unformatted`" pour des données non formatées.

Pour écrire à un tel fichier on a la primitive

```
write(nomUnite, donnees, nuRec, format);
```

où `donnees` est un tableau de données, `nuRec` est le numéro d'enregistrement dans le cas des fichiers en accès direct et `format` représente le format d'enregistrement. `format` doit être inclus dans des parenthèses, précédées et suivies d'un simple quote. Exemple : `'(1x,e10.3,5x,3(f3.0)) , (10x,a20)'`.

Pour la lecture on utilise la primitive

```
[donnees] = read(nomUnite, m, n, nuRec, format);
```

où `m, n` repésentent les dimension du tableau `donnees`. Si `m = -1`, alors la totalité du tableau `donnees` sera lue.

Avec un fichier nous pouvons aussi faire une actions selon la primitive :

```
file(action, nomUnite);
```

où action peut être :

- "close" : ferme le fichier correspondant.
- "rewind" : rebobine le fichier en plaçant le pointeur au début du fichier. Utile pour les fichiers en accès séquentiel.
- "backspace" : place le pointeur au début de l'enregistrement précédent. Utile pour les fichiers en accès séquentiel.
- "last" : place le pointeur après la fin du dernier enregistrement. Utile pour les fichiers en accès séquentiel.

Si nous voulons avoir une gestion selon le langage C, on doit utiliser pour l'ouverture des fichiers la primitive `mopen` (voir l'aide de Scilab).

Pour l'écriture sur un fichier on peut aussi utiliser la primitive `fprintf` ou si il s'agit d'afficher sur l'écran la primitive `printf`.

```
fprintf(nomUnite, format, val1, val2, ..., valN).
```

```
printf(format, val1, val2, ..., valN).
```

Le format est dans le style du format en langage C. Par exemple : `'val1 = %6.3f  val2 = %6.3f \n'`.

La primitive correspondante pour la lecture est fscanf et scanf pour la lecture des données sur clavier.

```
[v_1,...v_n]=fscanf (file,format);
```

```
[v_1,...v_n]=fscanf (format);
```

1.6. Exercices

EXERCICE 1.1 *Faire un programme qui permet d'évaluer la précision de l'ordinateur eps. Comparer avec la valeur* `%eps` *donnée par* `SciLab`.

EXERCICE 1.2 *Vérifier pour*

$$a = 10^{20}, \quad b = -10^{20}, \quad c = 1$$

que $(a+b)+c \neq a+(b+c)$. *Expliquer ce résultat.*

EXERCICE 1.3 *La recurrence*

$$x_{k+1} = 111 - (1130 - 3000/x_{k-1})/x_k, \quad x_0 = 11/2; \quad x_1 = 61/11$$

en arithmétique exacte converge vers la valeur 6.

Écrire un programme qui permet de calculer x_{34} *et comparer le résultat avec la vraie valeur qui est égale à 5.998. Expliquer la différence.*

EXERCICE 1.4 *Écrire un programme de résolution de l'équation du second degré* $ax^2 + bx + c =$ *et le tester sur les exemples suivants :*

(1) $a = 1, \quad b = -10^5, \quad c = 1$

(2) $a = 6, \quad b = 5, \quad c = -4$

(3) $a = 6 \cdot 10^{30}, \quad b = 5 \cdot 10^{30}, \quad c = -4 \cdot 10^{30}$

(4) $a = 10^{-30}, \quad b = -10^{30}, \quad c = 10^{30}$

Améliorer l'algorithme.

EXERCICE 1.5 *Soit la suite définie par*

$$x_n = \left(\frac{3b-a}{11}\right) 4^n + \left(\frac{12a-3b}{11}\right) \frac{1}{3^n}$$

avec $a = 1$ *et* $b = \frac{1}{3}$.

Calculer $x_1, x_2, \ldots, x_{50}$.

On définit maintenant la suite

$$x_n = \left(\frac{3b-a}{11} + \eta\right) 4^n + \left(\frac{12a-3b}{11} + \eta\right) \frac{1}{3^n}$$

avec $a = 1$, $b = \frac{1}{3}$ *et* $\eta = 10^{-16}$.

Calculer Calculer $y_1, y_2, \ldots, y_{50}$.

Expliquer les résultats.

1.7. Références

Pour la redaction de cette introduction à Scilab, nous avons utilisé les textes suivants :

- SCILAB GROUP : Introduction to Scilab, Inria
- G. E. URROZ : *Numerical and statistical methods with Scilab*, 2001
- B. PINÇON : Introduction au Scilab, `http ://www.iecn.u-nancy.fr/~pincon/`

Table des matières